1 Introduction

Packet switch networks are dynamically changing systems that transport packets of data to different sources to different destinations though a network of interconnected communication links. The state of communications links can change, becoming more or less congested or failing completely. To ensure robust data communication, we require efficient methods to choose routes across a network that can react quickly to communication link changes.

Route generation and selection algorithms for packet switch networks may be broadly categorized as either *optimal routing* algorithms or *shortest path* algorithm. Optimal routing algorithms determine network traffic flows to minimizes network wide cost that is a function of link delays. These algorithms are computationally intensive and often produce routes with several paths between a particular source and destination. Shortest paths algorithms determine a minimal cost between a particular source and destination. Short-path algorithms generally have polynomial complexity and generally only produce a produce a single path between a source and destination. Most packet-switched networks use some form of shortest-path algorithm to generate and select routes.

Shortest-path algorithms can be divided into two classes: *distance vector* and *link state*. Distance vector algorithms are based on dynamic programming models and can be implemented in a distributed, asynchronous framework using local cost estimates. The basic distance vector algorithm is known as the Bellman-ford algorithm or the Ford-Fulkerson method. Link state algorithms are usually implemented in a replicated fashion with each switch performing an independent route computation. To perform a route computation, link cost estimates are required for every link in the network. The basic link state method is Dijkstra's algorithm.

This set of lecture notes will present a brief overview of distance vector and link state link state routing, with emphasis on the Bellman-Ford and Dijkstra's algorithms. A more thorough treatment of routing with practical implementations can be found in [1, 2], with [1] being the more comprehensive source.

2 Shortest Path Routing

In shortest path routing, the topology communications network is represented using a directed weighted graph. The nodes in the graph represent switching elements and the directed arcs in the graph represent communication links between switching elements. Each arc has a weight that represents the cost of sending a packet between two nodes in a particular direction. This cost is generally a positive value that can inculcates such factors as delay, throughput, error rate, monetary cost etc. A path between two nodes may go through several intermediary nodes and arc. The objective in shortest path routing is to find a path between two nodes that has the smallest total cost, where the total cost of a path is the sum of the arc costs in that path.

3 Distance Vector Routing

Distance vector routing has been in practical use longer than link state algorithms, with the first large scale application in ARPANET [1, 2]. We represent the topology of a communication network with a weighted directed graph, where the nodes represent switching elements and links represent communication links. Let d_{ij} represent the cost of the link from node i to node j. If there is no link from node i to node j, then $d_{ij} = \infty$. In addition, let D_{ij} be the cost corresponding to the minimum-cost route or path from a source node i to a destination node j. Note that in the lecture slides, the destination node was assumed to be 1 and D_i was used to represent the shortest distance from source node i to the destination node 1. Assuming link costs are additive, the minimum-cost route from node i to node j can be found by solving Bellman's equation:

$$\begin{aligned} D_{ii} &= 0 \text{ for } i \\ D_{ij} &= \min_k \{d_{ik} + D_{kj}\} \text{ for } i \neq j \end{aligned} \tag{1}$$

The Bellman equation can be iteratively solved, using the Bellman-Ford algorithm as

$$\begin{aligned} D_{ii}^{k+1} &= 0, \text{ for } i \\ D_{ij}^{k+1} &= \min_l \{d_{il} + D_{lj}^k\}, \text{ for } i \neq j \end{aligned} \tag{2}$$

with initial condition $D_{ii}^0 = 0$ for all i and $D_{ij}^0 = \infty$ for $i \neq j$. The Bellman-Ford algorithm can be applied either for a fixed source node i or a fixed destination node j. In the lecture slides, the destination node is fixed, although other references [1, 2] use a fixed source node. We consider the case of a fixed destination node j for consistency with the lecture slides. For a fixed destination node j, the algorithm terminates when the minimum-cost do not change, $D_{ij}^{k+1} = D_{ij}^k$ for all i. The first iteration of the Bellman-Ford algorithm finds the shortest distance from all nodes to node j subject to the constraint that each path can have up to one link. On the second iteration, the maximum number of links in a path is constrained to two and on the k^{th} iteration, the maximum number of paths in each route is constrained to k. The Bellman-Ford algorithm hence takes the form of a dynamic program. For a network with N nodes, each iteration must be done for $N-1$ nodes where the minimization in (2) must be taken over a maximum of $N-1$. Since a path can have a maximum of $N-1$ links, Bellman-Ford algorithm will terminate after a maximum $N-1$ iteration, thus the computational the complexity of this algorithm is $O(N^3)$.

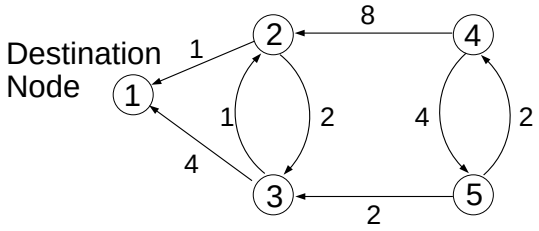

Figure 1: Network Graph - Bellman-Ford Algorithm

In order to demonstrate the workings of the Bellman-Ford algorithm, consider the network represented in Figure 1. The Bellman-Ford algorithm applied to the network in Figure 1 is shown in Table 1 for a destination node 1. The shortest distances do not change between the 4^{th} and 5^{th} iterations, thus the algorithm terminates. The paths to node 1 from all other nodes can be

represented in a minimum-cost spanning tree shown in Figure 2. The spanning tree indicated the path a packet would take to node 1. Note that if the direction of the arrows in Figure 1 were reversed, the entries in Table 1 would correspond to problem of finding the shortest distance from node 1 to all the other destination nodes.

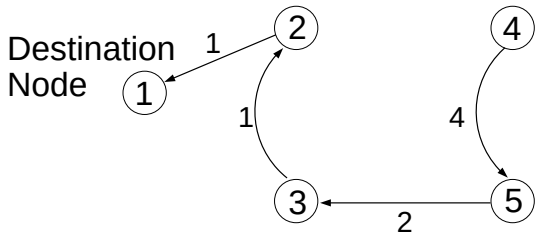

Figure 2: Spanning tree

Table 1: Iterations of Bellman-Ford Algorithm

Iteration #	$D_{1,1}$	$D_{2,1}$	$D_{3,1}$	$D_{4,1}$	$D_{5,1}$
0	0	∞	∞	∞	∞
1	0	1	4	∞	∞
2	0	1	2	9	6
3	0	1	2	9	4
4	0	1	2	8	4
5	0	1	2	8	4

The Bellman-Ford algorithm is well suited to communication network applications. The algorithm can be applied in a distributed fashion so that each node can calculate the shortest path to all other nodes. If one or more links should fail in a distributed implementation, it is possible for the nodes in the network to detect this failure and route around it. In a distributed implementation, each node i can estimate the cost to each destination j in the following manner. For each neighbor node k, node i collects k's estimates of the cost to route to j, adds d_{ik} to each estimate and selects the smallest one. The distributed Bellman-Ford algorithm hence only requires local cost information. For a given node, minimum distance information to each destination is usually stored in a *routing table*. Each entry in the routing table includes the address of $next - hop$ node on the way to the destination and distance or cost to the destination (hence the name *distance vector*). Practical implementation or the Bellman-Ford algorithm, such as Routing Information Protocol (RIP), also contain a timestamp on each routing table entry. For the i^{th} node, the next-hop router nh_{ij} to a destination node j is given by

$$\mathrm{nh}_{ij} = \{k^* : \{d_{ik^*} + D_{k^*j}\} = \min_k \{d_{ik} + D_{kj}\}\}, \quad \text{for } i \neq 1 \tag{3}$$

where the D_{kj}'s are the most recent minimum distance estimates from neighboring nodes. If a packet arrives at node i that is destined for node j, node i forwards this packet to node nh_{ij}.

Distributed implementations of the Bellman-Ford algorithm may be implemented synchronously

or asynchronously. The asynchronous Bellman-Ford algorithm is given by,

$$\begin{aligned} D_{ii}(t) &= 0, \text{ for } i \\ D^i_{ij}(t) &= \min_k\{d_{ik} + D^i_{kj}(t)\}, \text{ for } i \neq 1 \end{aligned} \tag{4}$$

where $D^k_{ij}(t)$ is node k's estimate of the cost from node i to node j at time t. The asynchronous Bellman-Ford algorithm will converge to the minimal-cost routes in finite time provided the following conditions hold [2]

1. Nodes must never stop recomputing or receiving cost estimates.
2. All cost estimates must be non-negative.
3. Old cost estimates do not remain indefinitely in the network.

and provided that there are link cost changes after a time t_0.

Thus far, we have not explicitly considered the effect of dynamic changes in link cost. Link cost changes generally result from either link failure or changes in link activity. A brute force solution to possible changes in link costs is to constantly update routing tables and compute minimum-cost routes. This can however waste a significant amount of network bandwidth. A better solution, which is uses in RIP implementations, is to trigger updates only when link costs change. Consider the case of a link cost change between node i and node nh_{ij}, where the node nh_{ij} detects a link cost change (this is typical since packets from i flow through nh_{ij} to j). The node nh_{ij} will indicate a link cost change to node i and node i will in turn indicate link cost changes to its other neighbors. If the link between nodes i and nh_{ij} fails, then a link cost change will not be triggered. To account for link failures, Bellman-Ford implementations allow for aging and invalidating of entries in the routing table. Periodically, a node has to ensure that it's links are valid and trigger updates if link failures are detected.

The Bellman-Ford algorithm is guaranteed to converge to the minimum-cost routs provided that the link cost do not change after a certain point in time. The algorithm will not converge however if link costs continually change and the convergence time is less than the time between successive changes. In some cases, inconsistency can arise that result in *routing loops* that may persist for some time.

Routing loops can occur as a result of the *count-to-infinity* problem. If a link between nodes i and j fails, then we assign an "infinite" cost to that link, which is in practice some number M that is larger than the cost of all other routes. For some neighbor node k of node i, the cost of the path from k-to-i-to-j is $M + d_{ki}$, but since M is the maximum routing cost, we assign a cost of M to this route. This can greatly slow down converges. Consider the network depicted in Figure 3, where for simplicity we have assumed bidirectional links with the same cost in both directions. Now assume that at some time t_0, node B detects a link failure in the link between B and D. For simplicity, assume a synchronous implementation of the asynchronous Bellman algorithm in (4) such that all nodes send updates at times t_i. The minimum distance routes for the Bellman-Ford algorithm at each time instant t_i are shown in Table 2 for a destination node D. Each entry in the table has a next-hop node and a distance to D. Note that '$*$' indicates a loop-back to A and '$-$' indicates an unreachable destination. At time t_0, B detects the failure of the link to D, but A, C still believe there is a path through C with cost of 2. At the next time instant, C fools B, A that there is a path through C to D with a cost of 3 even though no such path exists. C in turn believe there is a path through A with cost of 3. This routing loop continues until the costs equal the cost of using the link from C to D.

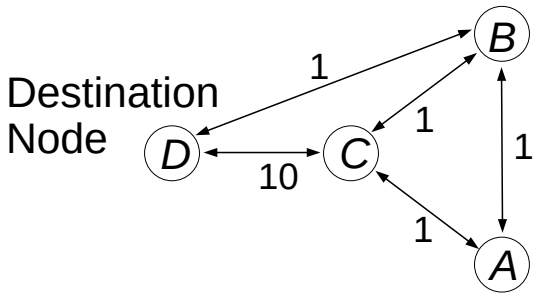

Figure 3: Network graph - count-to-infinity Problem

Table 2: Bellman-Ford Algorithm with Count-to-Infinity Problem

	t_0	t_1	t_2	t_3	$\cdots$	t_8	t_9
$D:$	$*, 0$	$*, 0$	$*, 0$	$*, 0$	$*, 0$	$*, 0$	
$B:$	$-$	$C, 3$	$C, 4$	$C, 5$	$C, 10$	$C, 11$	
$C:$	$B, 2$	$A, 3$	$A, 4$	$A, 5$	$A, 10$	$A, 10$	
$A:$	$B, 2$	$C, 3$	$C, 4$	$C, 5$	$C, 10$	$C, 11$	

The count-to-infinity problem can be readily solved using the technique of *Split horizon*. Split horizon does not pass cost updates to a neighbor if that neighbor is the next-hop to a particular destination. For example, at time t_1 the next-hop for node A is C, thus A can prevent a routing loop by not telling C there is a path to D through A.

4 Link State Routing

Link state routing was first applied in a mature implementation of ARPANET as a more flexible and robust alternative to distance-vector routing. In link-state routing, each node maintains a database of all the links in the network, hence the name "link-state". This database, know as the *link-state database*, is replicated across all the nodes in the networks. Each node can route packets to a particular destination by calculating the shortest path from itself to the destination node. The basic shortest path computation from a given node to all other nodes is performed using Dijkstra's algorithm.

Link-state routing over a communication network can be divided into two functions: computing shortest path routes and replicating link-state databases. The shortest path calculation is efficiently performed using Dijkstra's algorithm, at a computational cost that is less than Bellman-Ford based routing protocols. The task of replicating link-state information is accomplished by broadcasting *link-state advertisements* (LSA's), which contains the link-state data pertaining to a particular node. We will first describe Dijkstra's algorithm and then discuss the issues involved in distributing LSA's

Like the Bellman-Ford algorithm, Dijkstra's algorithm constructs a minimum weight spanning tree of a weighted direction of graph that represents the network topology. The root of this spanning tree is the source node. With each iteration, a new node is added to the tree such that the path cost to the new node is lower than the path cost to any other node not already in the tree. To formalize this procedure, we define a set P of *permanently labelled* nodes that is the set of nodes

already in the spanning tree. Similar to distance vector routing, we let d_{ij} represent the cost of the link from node i to node j and let D_{ij} be the minimum-cost path from a source node i to a destination node j. If there is is no link between a source and destination then $d_{ij} = \infty$ and if the source and destination are the same node then $d_{ii} = 0$. Dijkstra's algorithm for a source node i is a follows:

- Step 0: (Initialization) $P = \{i\}$, $D_{ij} = d_{ij}$, for all i
- Step 1: (Find closest node) Find $i \notin P$ such that

$$D_{ij} = \min_{k \notin P} D_{ik}$$

  Set $P = P \cup \{j\}$. If P contains all nodes, then stop.

- Step 2: (Update distance estimates) For all $k \notin P$ set

$$D_{ik} = \min\{D_{ik}, D_{ij} + d_{jk}\}$$

  Go to Step 1.

In the straight forward implementation of Dijkstra's algorithm, there are $N - 1$ iterations and each requires a number of operations that is proportionate to N, where N is the number of nodes. This leads to a computational complexity of $O(N^2)$, which is less that than the $O(N^3)$ complexity of the Bellman-Ford algorithm. If the set of nodes not in P are stored in a heap data structure, Dijkstra's algorithm can however be performed with complexity $O((N + l) \times \log(N))$, where l is the number of links in the network [2].

In order to demonstrate the workings of Dijkstra's algorithm, consider the network represented in Figure 4. For a source node 1, Dijkstra's algorithm has been applied to this network in Table 3. The i^{th} row indicates the distance from node 1 to each of the other node and also the nodes in the permanently labelled set after the i^{th} iteration. The bolded element indicates the node entering the permanently labelled set.

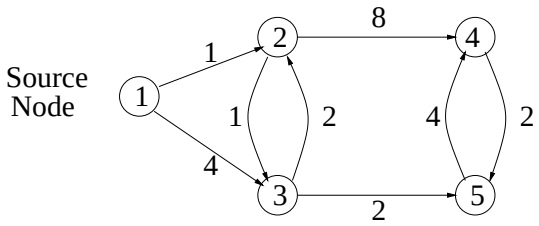

Figure 4: Network Graph - Dijkstra's algorithm

Effective link-state routing requires all switch to perform a shortest path calculation on a common link-state network graph. When a switch powers up, it obtains a copy of the common network graph from a neighboring switch. In the case of a link cost change, there needs to be a way to update and replicate the network graph. A brute force solution is to send the entire network link-state graph to all switches when the value of a link cost changes. This solution is not bandwidth efficient. A better solution is to only send an update of the part of the network graph that has changed. This is accomplished by representing the network graph as a set of LSA's that are stored

Table 3: Iterations of Dijkstra's Algorithm

Iteration #	$D_{1,1}$	$D_{1,2}$	$D_{1,3}$	$D_{1,4}$	$D_{1,5}$	P
init		**1**	4	∞	∞	1
1			**2**	9	∞	1, 2
2				9	**4**	1, 2, 3
3				**8**		1, 2, 3, 5
4						1, 2, 3, 5, 4

in a network database. Each LSA contains the link-state information pertaining to a particular switch, thus when a link cost changes, only the LSA of the affected switch needs to be sent to the other nodes. The LSA of a switch generally contain the following information:

- The identify of switch itself
- A list of operational links (outgoing links to forward traffic)
- Cost for each operational links
- Sequence number
- An indication of applications (mail,ftp,http) homed off the switch

A set of simplified LSA's for the nodes in Figure 4 is shown in Table 4.

Table 4: Link-state Advertisements for nodes in Figure 4

Switch 1		Switch 2		Switch 3		Switch 4		Switch 5	
$SW2$	1	$SW3$	3	$SW2$	2	$SW5$	2	$SW4$	4
$SW3$	4	$SW4$	8	$SW5$	2				

When a link cost changes, LSA's are broadcast to all the switches in a network through the method of *flooding*. Flooding of of LSA's can be accomplish in the following steps:

1. The switch receives a flooded LSA on one of its links
2. By examining sequence numbers, the switch determines if the received LSA is newer than the corresponding LSA in its database (if any). If the received LSA is newer, it replaces the current one and is rebroadcasted to all the switches links.
3. An acknowledgement is sent back on the link from which the LSA came from.

Flooding is a fast and robust way to ensure that all nodes receive an updated LSA. Accordingly, link-state algorithms react quickly to link cost changes and do not suffer from slow convergence issues that are present in distance-vector approaches.

5 Summary and Comparison

In this set of lecture notes, we presented two classes of shortest path algorithms: distance vector and link state. Both classes of routing algorithms can be implemented in a distributed asynchronous manner. Distance vector algorithms only require an exchange of local cost estimates, but can suffer from slow convergence and routing loops. Link state algorithms require global cost estimates, but are generally more robust and have faster convergence times.

References

[1] M. Steenstrup, *Routing in Communication networks*, Prentice Hall, New Jersey, 1995.

[2] D. Bertsekas, R. Gallager, *Data Networks*, Prentice Hall, New Jersey, 1987.